

Bash Shell Scripting for Helix and Biowulf

Dr. David Hoover, SCB, CIT, NIH

hoooverdm@helix.nih.gov

January 16, 2013

This Presentation Online

<http://helix.nih.gov/Documentation/Talks/BashScripting.pdf>

<http://helix.nih.gov/Documentation/Talks/BashScripting.tgz>



HISTORY AND INTRODUCTION

Bash

- Bash is a shell, like Bourne, Korn, and C
- Written and developed by the FSF in 1989
- Default shell for most Linux flavors

Definitive References

- [http://www.gnu.org/software/bash/
manual/](http://www.gnu.org/software/bash/manual/)
- <http://www.tldp.org/>
- <http://wiki.bash-hackers.org/>

Bash on Helix and Biowulf

- Helix runs RHEL v6, Bash v4.1.2
- Biowulf runs RHEL v5, Bash v3.2.25
- Subtle differences, 3.2.25 is a subset of 4.1.2
- This class deals with 3.2.25

Start

- Log onto Helix (or Biowulf)

```
$ ssh $USER@helix.nih.gov
```

- Create a scratch space to work in

```
$ mkdir -p /scratch/$USER/LINUXCLASS  
$ cd /scratch/$USER/LINUXCLASS
```

- Load LINUXCLASS module

```
$ module load LINUXCLASS
```

You might be using Bash already

```
$ ssh user@helix.nih.gov
...
Last login: Mon Jan  7 09:04:47 2013 from
dhcp528-241-99-230.abc.nih.gov
[user@helix ~]$ echo $SHELL
/bin/bash
```

If not, just start a new shell:

```
[user@helix ~]$ bash
[~]$ bash
```

What shell are you running?

- You should be running bash:

```
$ echo $0  
-bash
```

- Maybe you're running something else?

```
$ echo $0  
-ksh  
-csh  
-tcsh  
-zsh
```



ESSENTIALS

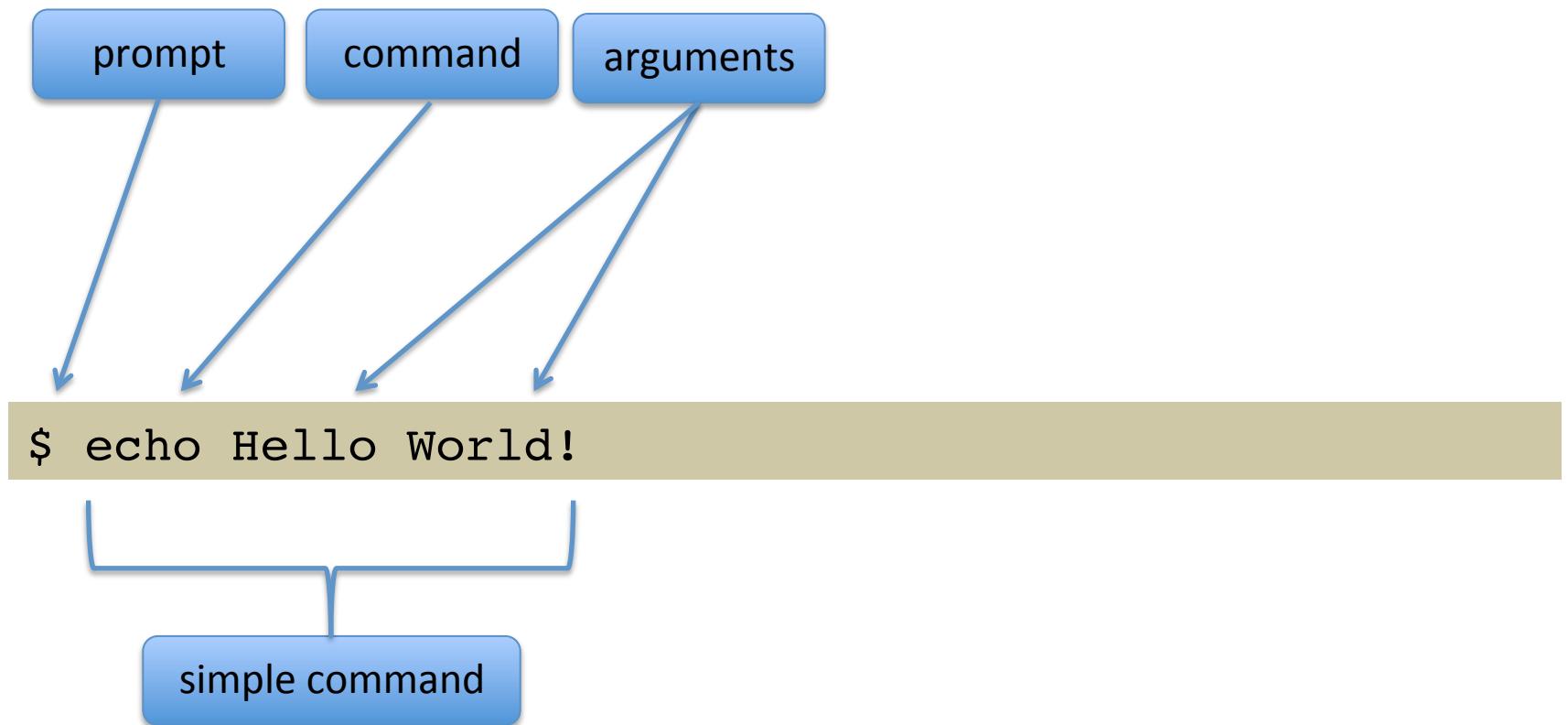
*nix Prerequisite

- It is essential that you know something about UNIX or Linux
- Introduction to Linux (Helix Systems)
- http://helix.nih.gov/Documentation/#_linuxtut
- <http://www.unixtutorial.org/>

Elemental Bash

- Bash is a command processor: interprets characters typed on the commandline and tells the kernel what programs to use and how to run them
- AKA command line interpreter (CLI)

Simple Command



*nix Commands



"wingardium leviOsa, not wingardium leviosA"

Essential *nix Commands

| cd | ls | pwd | file | wc | find | du |
|-------|-------|------------|------------|------------|------|-------|
| chmod | touch | mv | cp | rm | cut | paste |
| sort | split | cat | grep | head | tail | less |
| more | sed | awk | diff | comm | ln | |
| mkdir | rmdir | df | pushd | popd | | |
| date | exit | ssh | rsh | printenv | time | echo |
| ps | jobs | [CTRL-C] | [CTRL-Z] | [CTRL-D] | top | kill |
| type | help | man | apropos | | | |

type

- type is a builtin that displays the type of a word

```
$ type -t rmdir  
file  
$ type -t if  
keyword  
$ type -t echo  
builtin  
$ type -t module  
function  
$ type -t ls  
alias
```

Command Documentation

```
$ man pwd  
$ man diff  
$ man head
```

builtin

- Bash has built-in commands (`builtin`)
- Documentation can be seen with `help`

```
$ help
```

- Specifics with `help [builtin]`

```
$ help pwd
```

Non-Bash Commands

- Found in `/bin`, `/usr/bin`, and `/usr/local/bin`
- Some overlap between Bash builtin and external executables

```
$ help time  
$ man time
```

```
$ help pwd  
$ man pwd
```



SETTING THE ENVIRONMENT

Environment

- A set of variables recognized by the kernel and used by most programs
- Not all variables are environment variables, must be **exported**
- Initially set by **startup files**
- **printenv** displays variables and values

Set a variable

- Very simple, no spaces

```
$ myvar=10
```

- Examine value with echo and \$:

```
$ echo $myvar  
10
```

- This is actually **parameter expansion**

Set a variable

- Formally done using `declare`

```
$ declare myvar=100
```

- Can set variable to read-only

```
$ declare -r myvar="secret word"  
$ myvar="replacement"  
-bash: myvar: readonly variable
```

- Or using `readonly`

```
$ readonly myvar
```

export

- Export is used to set an environment variable:

```
$ MYENVVAR=10  
$ export MYENVVAR  
$ printenv MYENVVAR
```

- You can do it one move

```
$ export MYENVVAR=10
```

Unset a variable

- Unset is used for this

```
$ unset myvar
```

- Can be used for environment variables

```
$ echo $HOME  
/home/user  
$ unset HOME  
$ echo HOME  
  
$ export HOME=/home/user
```

- Read-only variable can not be unset

Bash Variables

- \$HOME = /home/[user] = ~
- \$PWD = current working directory
- \$PATH = list of filepaths to look for commands
- \$CDPATH = list of filepaths to look for directories
- \$TMPDIR = temporary directory (/tmp)
- \$RANDOM = random number
- Many, many others...

printenv

```
$ printenv
HOSTNAME=helix.nih.gov
TERM=xterm
SHELL=/bin/bash
HISTSIZE=500
SSH_CLIENT=96.231.6.99 52018 22
SSH_TTY=/dev/pts/274
HISTFILESIZE=500
USER=student1
```

module

- module can set your environment

```
$ module load python/2.7.3  
$ module unload python/2.7.3
```

- Can be used for environment variables

```
$ module avail
```

[http://helix.nih.gov/Applications/
modules.html](http://helix.nih.gov/Applications/modules.html)



LOGIN

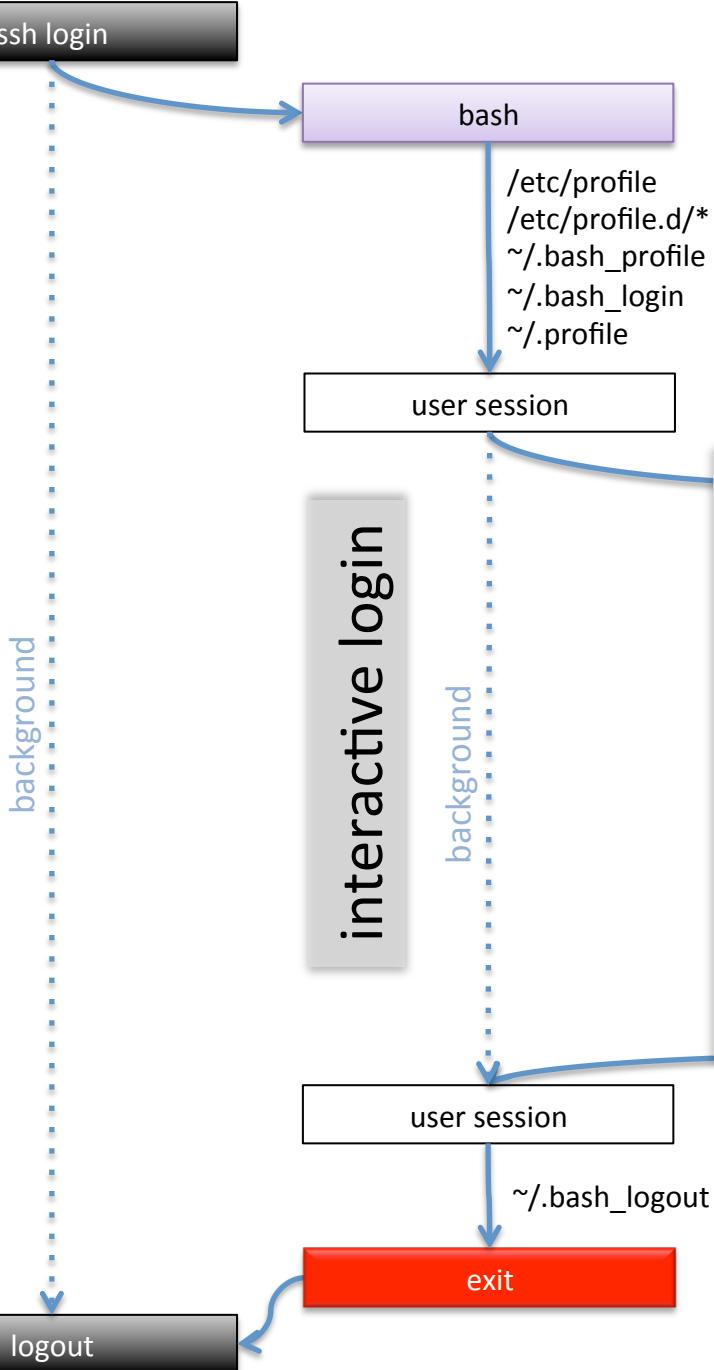
Logging In

- ssh is the default login client

```
$ ssh $USER@helix.nih.gov
```

- what happens next?

Bash Flow



Logging In

- Interactive login shell (ssh from somewhere else)

/etc/profile (/etc/bashrc?)
~/.bash_profile

~/.bash_logout (when exiting)

- The startup files are sourced, not executed

~/.bash_profile

```
$ cat ~/.bash_profile
# .bash_profile

# Get the aliases and functions
if [ -f ~/.bashrc ]; then
    . ~/.bashrc
fi

# User specific environment and startup programs

PATH=$PATH:$HOME/bin

export PATH
```

Non-Login Shell

- Interactive non-login shell (calling bash from the commandline)
- Retains environment from login shell
 - ~/.bashrc
- Shell levels seen with \$SHLVL

```
$ echo $SHLVL  
1  
$ bash  
$ echo $SHLVL  
2
```

Non-Interactive Shell

- From a script
- Retains environment from login shell

`$BASH_ENV` (if set)

- Set to a file like `~/.bashrc`

Difference between `source`, `exec`, and `. /`

- `source` runs commands in the current shell, and retains the results
- `. /` runs commands in a child shell, then returns with no changes to the parent shell
- `exec` displaces the current shell with a new shell and runs the command

Aliases

- A few aliases are set by default

```
$ alias
alias l.='ls -d .* --color=auto'
alias ll='ls -l --color=auto'
alias ls='ls --color=auto'
alias vi='vim'
```

- Can be added or deleted

```
$ unalias ls
$ alias ls='ls -CF'
```

- Remove all aliases

```
$ unalias -a
```

Functions

- functions are a defined set of commands assigned to a word

```
$ function status() {  
> date  
> uptime  
> who | grep $USER  
> checkquota  
> }  
$ status  
Thu Oct 18 14:06:09 EDT 2012  
    14:06:09 up 51 days, 7:54, 271 users, load average: 1.12, 0.91, 0.86  
user pts/128      2012-10-17 10:52 (128.231.77.30)  
Mount           Used     Quota   Percent   Files   Limit  
/data:          92.7 GB  100.0 GB  92.72%  233046  6225917  
/home:          2.2 GB   8.0 GB   27.48%  5510      n/a
```

Functions

- functions can propagate to child shells using `export`

```
$ export -f status
```

Functions

- `unset` deletes function

```
$ unset status
```

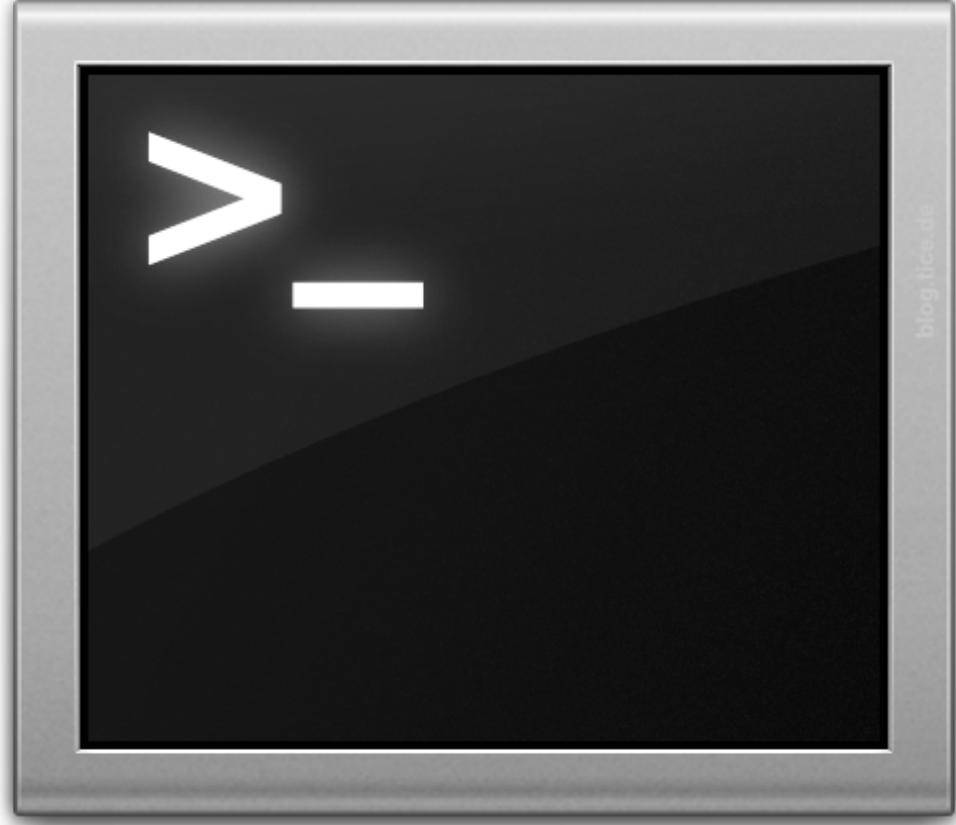
- use `declare -f` to display code

```
$ declare -f 'status'  
status ()  
{  
    date;  
    uptime;  
    who | grep --color $USER;  
    checkquota  
}
```

Functions

- local variables can be set using local

```
$ export TMPDIR=/tmp
$ function processFile() {
> local TMPDIR=/data/user/tmpdir
> echo $TMPDIR
> sort $1 | grep $2 > $2.out
> }
$ processFile /path/to/file string
/data/user/tmpdir
$ echo $TMPDIR
/tmp
```



blog.tice.de

COMMAND LINE INTERPRETER

Watch What You Write

- Bash interprets what you write *after* you hit return
- Patterns of characters can cause **expansion**

Parameter Expansion

- \$ is placed outside for parameter expansion

```
$ name=monster  
$ echo $name  
monster
```

- Braces can be used to preserve variable

```
$ echo $name_silly  
  
$ echo ${name}_silly  
monster_silly
```

Parameter Expansion

- More than one in a row

```
$ var1=cookie
$ var3=_is_silly
$ echo ${var1}_${name}${var3}
cookie_monster_is_silly
```

Brace Expansions

- Brace expansion { , , }

```
$ echo {bilbo,frodo,gandalf}  
bilbo frodo gandalf  
$ echo {0,1,2,3,4,5,6,7,8,9}  
0 1 2 3 4 5 6 7 8 9
```

- Brace expansion { .. }

```
$ echo {0..9}  
0 1 2 3 4 5 6 7 8 9  
$ echo {bilbo..gandalf}  
{bilbo..gandalf}  
$ echo {b..g}  
b c d e f g
```

Brace Expansions

- Nested brace expansions

```
$ mkdir z{0,1,2,3,4,5,6,7,8,9}
$ ls
z0 z1 z2 z3 z4 z5 z6 z7 z8 z9
$ rmdir z{{1..4},7,8}
$ ls
z0 z5 z6 z9
```

- Distinct from parameter expansion (\$ or \${ })

```
$ echo ${var1,${name}},brought,to,you,by,{1..3}
cookie monster brought to you by 1 2 3
```

Arithmetic Expansion

- `(())` is used to evaluate math
- `$` is placed outside for parameter expansion

```
$ echo ((12-7))
```

```
$ echo $((12-7))  
5
```

Arithmetic Expansion

- Variables can be updated, not just evaluated

```
$ a=4
$ b=8
$ echo $((a+b))
12
$ echo $a
4
$ echo $b
8
$ echo $( (a=a+b) )
12
$ echo $a
12
```

Arithmetic Expansion

- The ++ and -- operators only work on variables, and update the value

```
$ a=4
$ ((a++))
$ echo $a
5
$ unset b
$ ((b--))
$ echo $b
-1
$ ((4++))
-bash: 4++: syntax error: operand expected (error token is
"+")
```

Arithmetic – integers only

- Bash can only handle integers

```
$ a=4.5
$ ((a=a/3))
-bash: ((: 4.5: syntax error: invalid arithmetic operator
(error token is ".5")
```

Arithmetic – integers only

- Bash can only do integer math

```
$ a=3  
$ ((a=a/7))  
$ echo $a  
0
```

- Division by zero is caught with exit status

```
$ ((a=a/0))  
-bash: let: a=a/0: division by 0 (error token is "0")
```

Arithmetic Expansion

- Math is done using let and ‘(())’

```
$ a=1
$ echo $a
1
$ let a++
$ echo $a
2
$ ((a++))
$ echo $a
3
$ let a=a+4
$ echo $a
7
```

Other Expansions

- Command substitution

```
$ echo uname -n  
uname -n  
$ echo `uname -n`  
biowulf.nih.gov
```

Other Expansions

- Tab expansion

```
$ ls /usr/ ←  
bin/          lib/          local/          sbin/          share/  
$ ls /usr/
```



hit tab here

Other Expansions

- Tilde expansion (~)

```
$ echo ~  
/home/user
```

Pattern Matching

- * : match any string
- ? : match any single character
- [?-?] : match range of characters
- [!?] or [^?] : not match character

```
$ touch {{1..9},{a..z}}
$ ls [a-e1-4]
1  2  3  4  a  b  c  d  e
$
```

Character Classes

- `[:CLASS:]` can be included with pattern matching

```
$ touch {1..9} y{1..9} z{1..9}  
$ ls [:alpha:]4  
y4 z4
```

- only on Helix (bash v4.1.2)

| | | | |
|-------|-------|-------|--------|
| alnum | cntrl | print | word |
| alpha | digit | punct | xdigit |
| ascii | graph | space | |
| blank | lower | upper | |

Quotes

- Single quotes preserve literal values

```
$ echo 'cd $PWD `uname -n`'  
cd $PWD `uname -n`
```

- Double quotes allow variable and shell expansions

```
$ echo "cd $PWD `uname -n`"  
cd /home/user helix.nih.gov
```

Quotes

- Double quotes also preserve blank characters

```
$ msg=`echo Hi$'\t'there.$'\n'How are you?`  
$ echo $msg  
Hi there. How are you?  
$ echo "$msg"  
Hi      there.  
How are you?
```

tab, not space

Escapes

- The escape character ‘\’ preserves literal value of following character

```
$ echo \$PWD is $PWD  
$PWD is /home/user
```

- However, it treats newline as line continuation

```
$ echo \$PWD is $PWD \  
> something else  
$PWD is /home/user something else
```

Escapes

- Funny non-printing characters

```
$ echo Hello World  
Hello World  
$ echo $'\n\n'Hello$'\t'World$'\n\n'
```

```
Hello    World
```

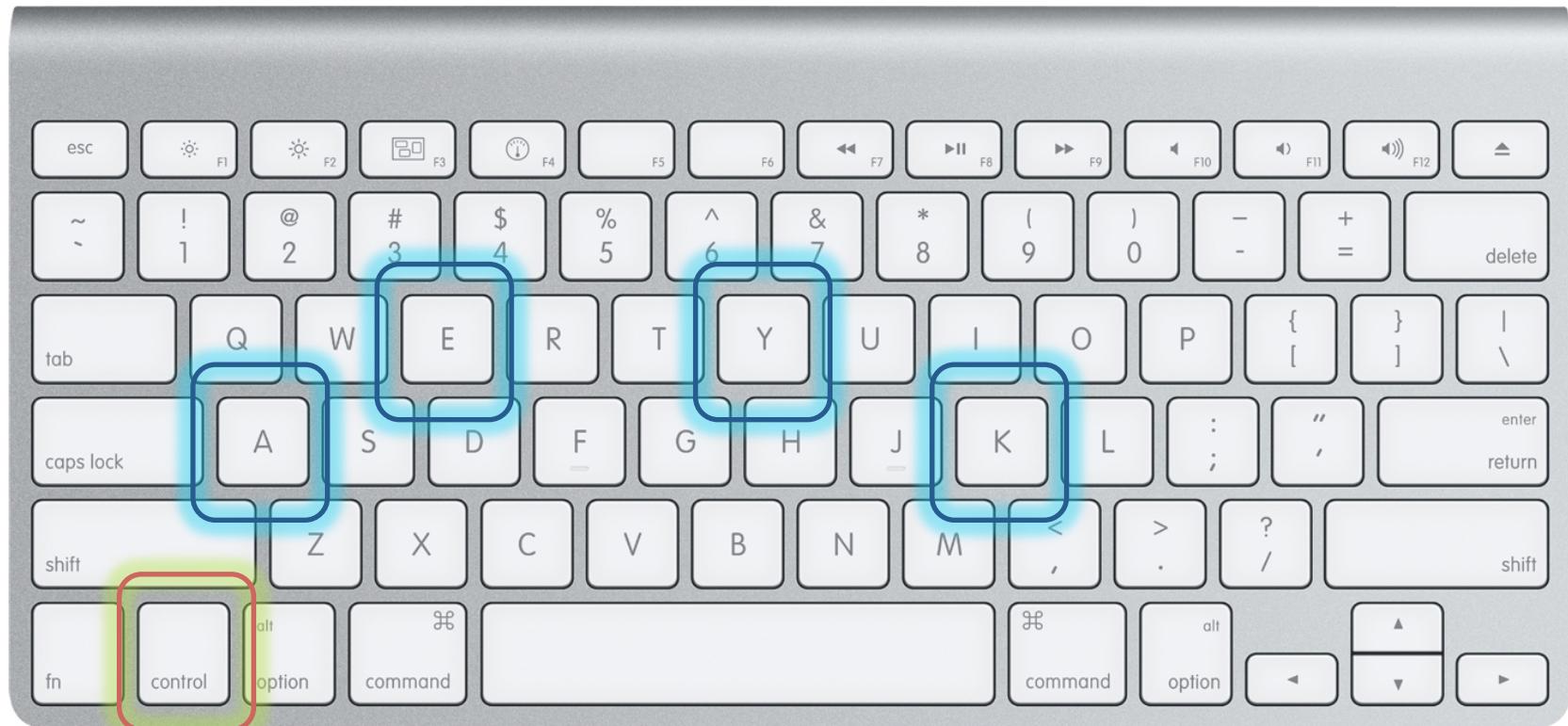
Readline

- Readline is a library of commands and variables (called keybindings) that control how you interact with the commandline
- Handled through bind (help bind)
- Some features include:
 - Alert bell
 - Cutting and pasting text
 - Setting comment character
 - Controlling length of history

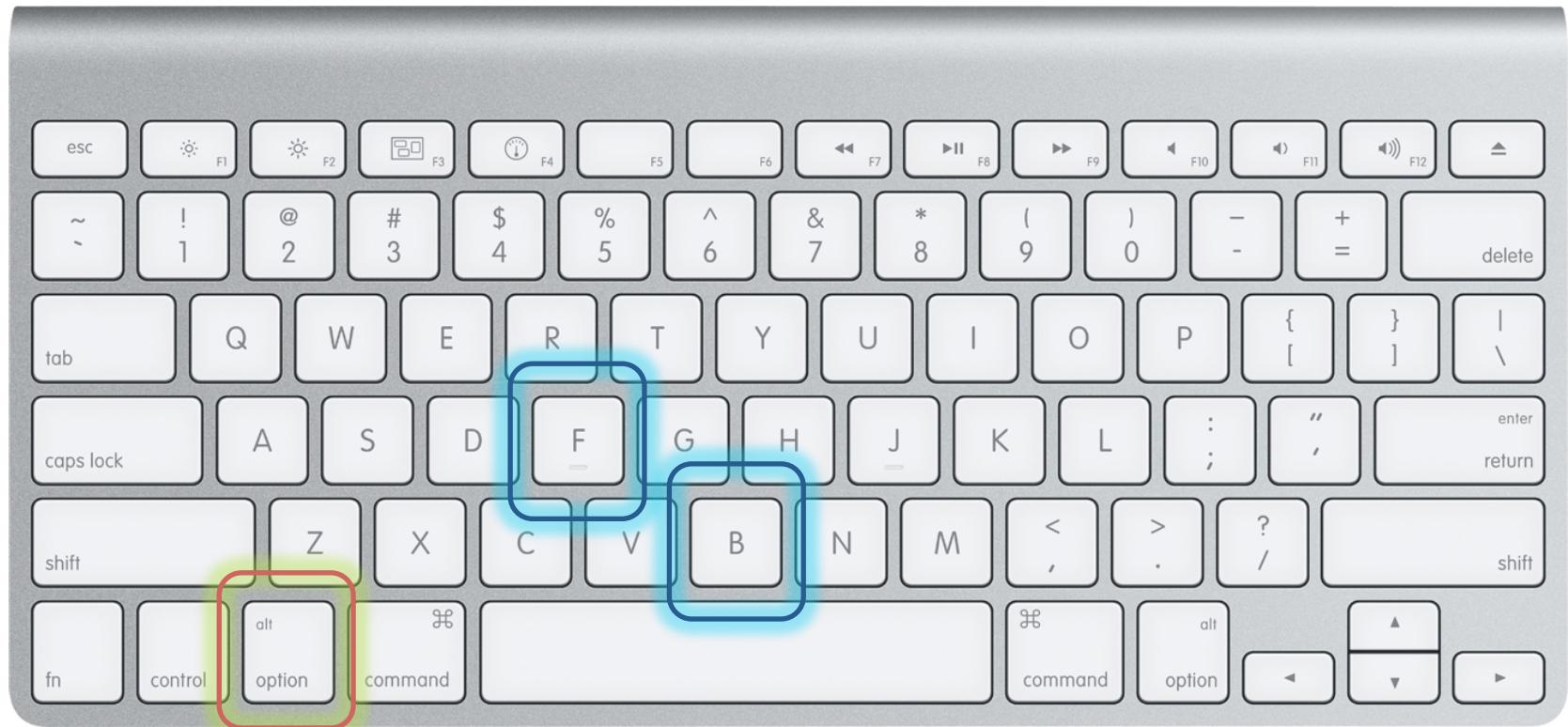
Readline Examples

- Ctrl-a : move to start of commandline
- Ctrl-e : move to end of commandline
- Meta-f : move forward one word
- Meta-b : move backward one word
- Ctrl-k : kill (cut) to end of line
- Ctrl-y : yank (paste) cut text at cursor
- bind -P to show all possibilities

Ctrl-a, Ctrl-e, Ctrl-k, Ctrl-y



Meta-f, Meta-b



Need to enable option as meta key, see Terminal -> Preferences...

History

- By default Bash keeps a history of commands given
- Generally arrow-up and arrow-down allow walking through previous commands
- More complete control is possible



<http://www.gnu.org/software/bash/manual/bashref.html#Using-History-Interactively>



SCRIPTS

Why write a script?

- One-liners are not enough
- Quick and dirty prototypes
- Maintain library of functional tools
- Glue to string other apps together

Why NOT to use a script?

- *Complex* math (beyond integers)
- *Complex* I/O (sockets, rsh, tcp)
- *Complex* data structures (hashes, multidimensional arrays)
- *Complex* file operations
- *Complex* pattern matching

Hardcore *nix Editors

- vi, nano, pico, emacs

```
$ vi script.sh  
$ bash script.sh
```

- Shebang for executable scripts

```
$ head -1 script.sh  
#!/bin/bash  
$ chmod +x script.sh  
$ ./script.sh
```

Desktop Access

- Mount /home or /data on your desktop and use a local program
- [http://helix.nih.gov/Documentation/
transfer.html](http://helix.nih.gov/Documentation/transfer.html)
- Most desktops and desktop support personnel regard scripts as a *threat*

Lightweight Editing Files

- Most reliable method:

```
$ echo '#!/bin/bash' > script1.sh
```

- On Macs, open withTextEdit
- On Windows, open with Notepad – run dos2unix after saving

```
$ dos2unix script1.sh
```

- Change permissions and run

```
$ chmod +x script1.sh  
$ ./script1.sh
```

Contextual Script Editors

- Script editors are very, very helpful
- Mac: Tincta
- Windows: FreeScriptEditor, Notepad++
- Linux: SciTE
- Proprietary (\$\$) editors probably better
- Resistance is futile. You will be assimilated.



Debugging

- Call bash with `-xv`

```
$ head -1 script.sh  
#!/bin/bash -xv
```

- Will display each active line, along with results and other information

Special Parameters - Positional

- Positional parameters are arguments passed to the shell when invoked
- Denoted by \${digit}, > 0

```
$ cat x.sh
echo $1 $2 $3 $4 $5 $6 $7 $8 $9 ${10} ${11} ${12}
$ bash x.sh {A..Z}
A B C D E F G H I J K L
```

- Normally used with scripts and functions

Shell Parameters - Special

Special parameters have a single character

\$* expands to the positional parameters

\$@ virtually the same as \$*

\$# number of positional parameters

\$- current option flags when shell invoked

\$\$ process id of the shell

\$! process id of last executed background command

\$0 name of the shell or shell script

\$_ final argument of last executed foreground command

\$? exit status of last executed foreground command

Sample Script

```
#!/bin/bash
module load R/2.14
cd /data/user
for i in {1..22} X Y M ; do
    label=$i
    if [[ $i == [:digit:] ]]; then
        label=`printf "%02d" $i`
    fi
    test -f x/$label/trial.out && break
    test -d x/$label || mkdir -p x/$label
    pushd x/$label 2>&1 > /dev/null
    echo Running chr${i}_out
    # actually do something, not this
    touch trial_chr${i}.out
    popd >& /dev/null
done
```



SIMPLE COMMANDS

Definitions

Command

word

Simple command

command arg1 arg2 ...

Pipeline / Job

simple command 1 | simple command 2 | & ...

List

pipeline 1 ; pipeline 2 ; ...

Some command examples

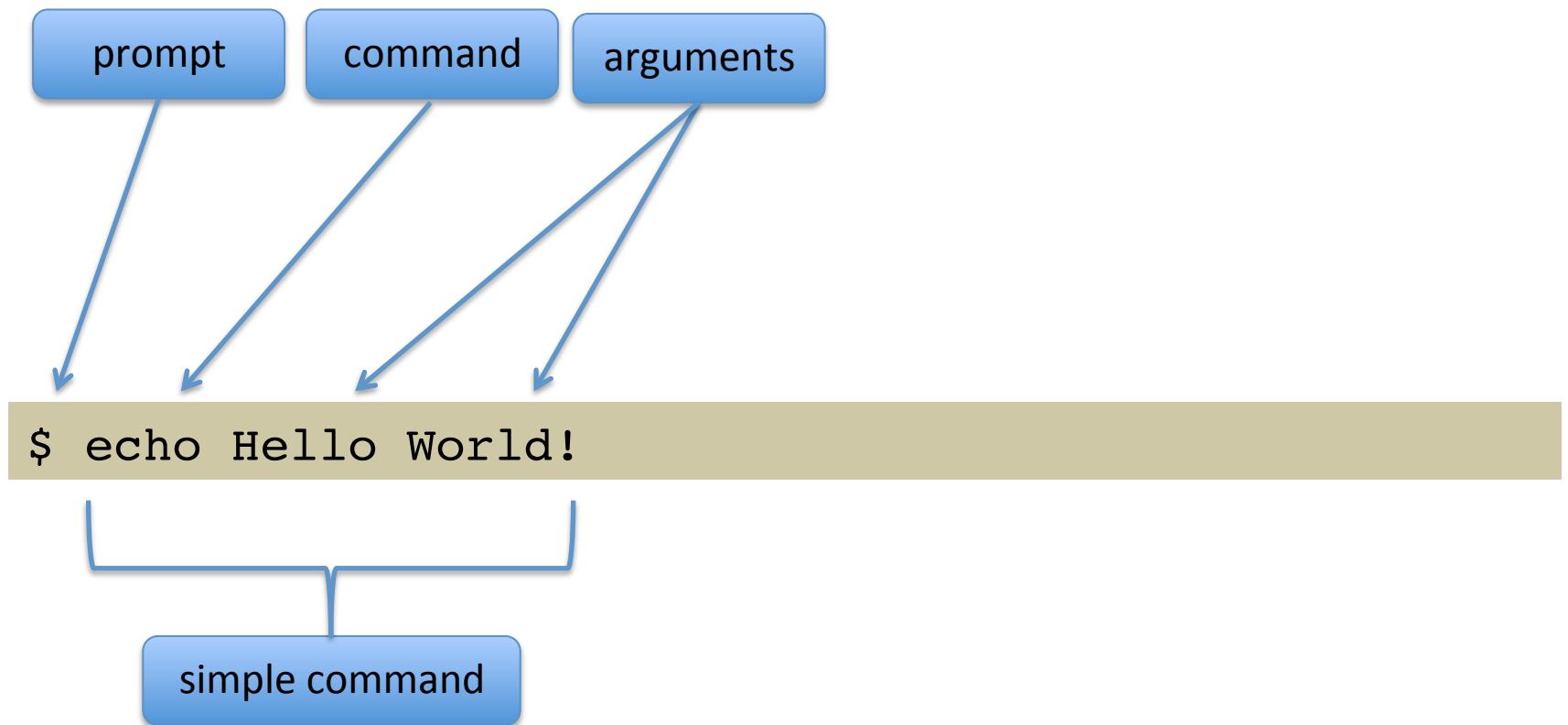
- What is the current time and date?

```
$ date
```

- Where are you?

```
$ pwd
```

Simple Command



Simple commands

- List the contents of your /home directory

```
$ ls -l -a $HOME
```

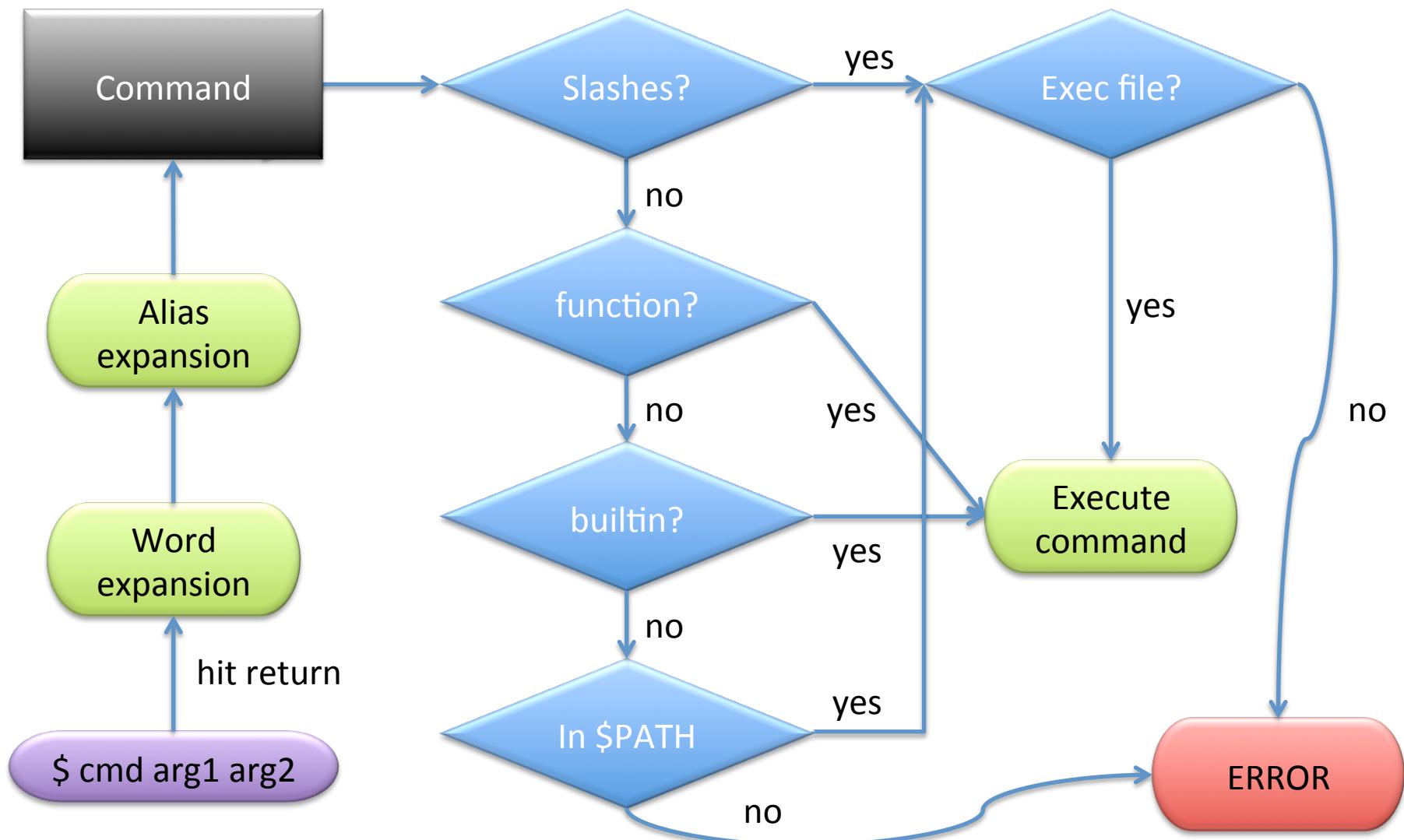
- How much disk space are you using?

```
$ du -h -s $HOME
```

- What are you up to?

```
$ ps -u $USER -o size,pcpu,etime,comm --forest
```

Command Search Tree



Process

- A *process* is an executing instance of a simple command
- Can be seen using ps command
- Has a unique id (*process id*, or *pid*)
- Belongs to a process group

top command

```
top - 15:51:30 up 5 days, 19:16, 240 users, load average: 15.40, 14.51, 14.77
Tasks: 4930 total, 16 running, 4897 sleeping, 17 stopped, 0 zombie
Cpu(s): 5.0%us, 1.6%sy, 3.9%ni, 89.4%id, 0.0%wa, 0.0%hi, 0.1%si, 0.0%st
Mem: 1058786896k total, 969744396k used, 89042500k free, 87800k buffers
Swap: 67108856k total, 2736k used, 67106120k free, 650786452k cached
```

| PID | USER | PR | NI | VIRT | RES | SHR | S | %CPU | %MEM | TIME+ | COMMAND |
|--------|----------|----|----|-------|------|------|---|-------|------|-----------|-----------------|
| 77108 | wenxiao | 39 | 19 | 63.8g | 63g | 1696 | R | 100.0 | 6.3 | 104:15.91 | getAlignmentSta |
| 98656 | guptaas | 39 | 19 | 4868m | 1.2g | 18m | R | 100.5 | 0.1 | 1287:38 | MathKernel |
| 254799 | wenxiao | 20 | 0 | 13.1g | 13g | 1664 | R | 100.0 | 1.3 | 3:48.60 | getAlignmentSta |
| 52986 | wenxiao | 39 | 19 | 77.0g | 76g | 1696 | R | 100.0 | 7.6 | 104:19.32 | getAlignmentSta |
| 59585 | lobkovsa | 39 | 19 | 24616 | 11m | 1136 | R | 100.0 | 0.0 | 193:29.44 | org_level_corr_ |
| 60824 | hex2 | 39 | 19 | 28.9g | 28g | 4100 | R | 100.0 | 2.8 | 7483:20 | R |
| 245039 | lobkovsa | 20 | 0 | 24616 | 11m | 1136 | R | 100.0 | 0.0 | 29:18.36 | org_level_corr_ |
| 245092 | lobkovsa | 20 | 0 | 24616 | 11m | 1136 | R | 100.0 | 0.0 | 29:06.75 | org_level_corr_ |
| 254829 | wenxiao | 20 | 0 | 12.1g | 11g | 1664 | R | 100.0 | 1.1 | 3:39.88 | getAlignmentSta |
| 10184 | rdmorris | 20 | 0 | 531m | 282m | 4640 | R | 99.9 | 0.0 | 0:43.47 | R |
| 245080 | lobkovsa | 20 | 0 | 24616 | 11m | 1136 | R | 99.6 | 0.0 | 29:13.30 | org_level_corr_ |
| 252981 | javiergc | 20 | 0 | 9204m | 465m | 55m | S | 99.6 | 0.0 | 9:12.76 | MATLAB |
| 179412 | sedavis | 39 | 19 | 63624 | 7632 | 2876 | S | 11.3 | 0.0 | 442:10.21 | ssh |

ps command

```
[root@helix ~]# ps -u rdmorris -f --forest
UID      PID  PPID  C STIME TTY          TIME CMD
rdmorris 242197 242128  0 Dec12 ?        00:00:00 sshd: rdmorris@pts/107
rdmorris 242198 242197  0 Dec12 pts/107  00:00:00  \_ -bash
rdmorris 114343 114296  0 Dec12 ?        00:00:02 sshd: rdmorris@pts/251
rdmorris 114344 114343  0 Dec12 pts/251  00:00:00  \_ -bash
rdmorris 55737 114344  0 12:26 pts/251  00:00:00      \_ /bin/bash /home/rdmorris/MascotTools/M...
rdmorris 55738 55737 10 12:26 pts/251  00:22:01      \_ /usr/local/R-2.13-64/lib64/R/bin/e...
rdmorris 46893 46840  0 Dec12 ?        00:00:09 sshd: rdmorris@pts/112
rdmorris 46915 46893  0 Dec12 pts/112  00:00:00  \_ -bash
```

Exit Status

- A process returns an exit status (0-255)
- 0 = success (almost always)
- 1 = general error, 2-255 = specific error
- Stored in \$? parameter

```
$ cat /var/audit
cat: /var/audit: Permission denied
$ echo $?
1
$ ls /zzz
ls: cannot access /zzz: No such file or directory
$ echo $?
2
```

Redirection

- Every process has three file descriptors (file handles): STDIN (0), STDOUT (1), STDERR (2)
- Content can be redirected

```
cmd < x.in
```

Redirect file descriptor 0 from STDIN to x.in

```
cmd > x.out
```

Redirect file descriptor 1 from STDOUT to x.out

```
cmd 1> x.out 2> x.err
```

Redirect file descriptor 1 from STDOUT to x.out,
file descriptor 2 from STDERR to x.err

Combine STDOUT and STDERR

```
cmd 2>&1
```

Redirect file descriptor 2 from STDERR to wherever file descriptor 1 is pointing (STDOUT)

- Ordering is important

Correct:

```
cmd > x.out 2>&1
```

Redirect file descriptor 1 from STDOUT to filename x.out, then redirect file descriptor 2 from STDERR to wherever file descriptor 1 is pointing (x.out)

Incorrect:

```
cmd 2>&1 > x.out
```

Redirect file descriptor 2 from STDERR to wherever file descriptor 1 is pointing (STDOUT), then redirect file descriptor 1 from STDOUT to filename x.out

Redirection to a File

- Use better syntax instead – these all do the same thing:

```
cmd > x.out 2>&1
```

```
cmd 1> x.out 2> x.out
```

WRONG!

```
cmd &> x.out
```

```
cmd >& x.out
```

Redirection

- Appending to a file

```
cmd >> x.out
```

Append STDOUT to x.out

```
cmd 1>> x.out 2>&1
```

Combine STDOUT and STDERR, append to x.out

```
cmd &>> x.out
```

Only available on bash v4 and above (Helix, not Biowulf)

~~```
cmd >>& x.out
```~~

WRONG!

# Named Pipes

- Send STDOUT and/or STDERR into temporary file for commands that can't accept ordinary pipes

```
$ ls /home/$USER > file1_out
$ ls /home/$USER/.snapshot/Weekly.2012-12-30* >
file2_out
$ diff file1_out file2_out > diff.out
$ rm file1_out file2_out
$ cat diff.out
```

# Named Pipes

- FIFO special file can simplify this
- You typically need multiple sessions or shells to use named pipes

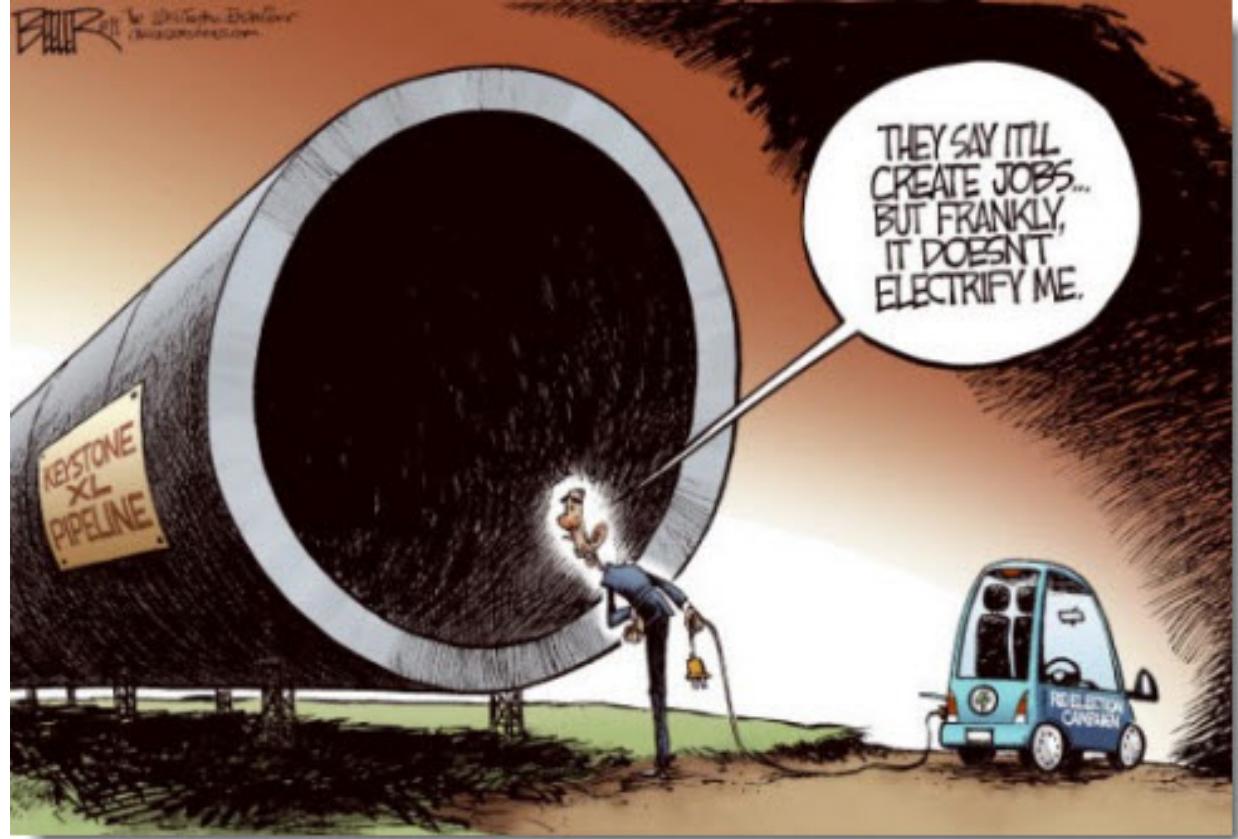
```
[sess1]$ mkfifo pipe1
[sess1]$ ls /home/$USER > pipe1
```

```
[sess2]$ cat pipe1
```

# Named Pipes

- The operator <( ) can be used to create transient named pipes

```
$ diff <(ls /home/$USER) <(ls /home/$USER/.snapshot/
Weekly.2012-12-30*)
```



# PIPELINES AND JOBS

# Pipeline

- STDOUT of each simple command is passed as STDIN of the next command

```
$ ps -ef | grep sh | head -3
```

# Pipeline

- STDERR can be combined with STDOUT

```
$ ls emptydir | grep -c 'No such file'
ls: cannot access emptydir: No such file or
directory
0
$ ls emptydir 2>&1 | grep -c 'No such file'
1
```

# Job Control

- A *job* is another name for pipeline

```
$ echo Hello World! > x | cat x | grep o
```

- Each simple command is a process

# Foreground and Background

- A job (pipeline) runs in the foreground by default
- *Asynchronous* jobs are run in background (in parallel, fire and forget)

```
$ sleep 5 &
```

- Requires single ' & ' at the end
- Background jobs run in their own shell
- Exit status is not available

# Common Problem

- A background process without redirection will die if the terminal is disconnected and the process tries to write to STDIN or STDOUT

```
$ STDIN_FAIL.sh &
```

- Make sure all background processes redirect STDIN and STDOUT

```
$ STDIN_FAIL.sh &> junk.out &
```

# Job Control

- The shell itemizes jobs by number

```
$ sleep 10 &
[1] 8683
$ sleep 10 &
[2] 8684
$ sleep 10 &
[3] 8686
$ jobs
[1] Running sleep 10 &
[2]- Running sleep 10 &
[3]+ Running sleep 10 &
$
[1] Done sleep 10
[2]- Done sleep 10
[3]+ Done sleep 10
```

# Job Control

- A job can be moved from foreground to background with `ctrl-z` and `bg`

```
$ step1.sh | step2.sh | grep normal
ctrl-z
[1]+ Stopped step1.sh | step2.sh ...
bg
[1]+ step1.sh | step2.sh ...
```

- Can be brought back with `fg`

```
$ jobs
[1]+ Running step1.sh | step2.. &
$ fg
step1.sh step2.sh ...
```



# COMMAND LISTS

# Command List

- Sequence of one or more jobs separated by ‘;’, ‘&’, ‘&&’, or ‘| |’.
- Simple commands/pipelines/jobs are run sequentially when separated by ‘;’

```
$ echo 1 > x ; echo 2 > y ; echo 3 > z
```

```
$ date ; sleep 5 ; sleep 5 ; sleep 5 ; date
```

# Command List

- Sequential command list is equivalent to

```
$ echo 1 > x
$ echo 2 > y
$ echo 3 > z
```

- or

```
$ date
$ sleep 5
$ sleep 5
$ sleep 5
$ date
```

# Command List

- Asynchronously when separated by ‘&’

```
$ echo 1 > x & echo 2 > y & echo 3 > z &
```

- wait pauses until all background jobs complete

```
$ date ; sleep 5 & sleep 5 & sleep 5 & wait ; date
```

# Command List

- Asynchronous command list is equivalent to

```
$ echo 1 > x &
$ echo 2 > y &
$ echo 3 > z &
```

- or

```
$ date
$ sleep 5 &
$ sleep 5 &
$ sleep 5 &
$ wait
$ date
```

# Grouped Command List

- To execute a list of sequential pipelines in the background, or to pool STDOUT/STDERR, enclose with ‘( )’ or ‘{ }’

```
$ (cmd 1 < input ; cmd 2 < input) > output &
[1] 12345
$ { cmd 1 < input ; cmd 2 < input ; } > output &
[2] 12346
```

- STDIN is NOT pooled, but must be redirected with each command or pipeline.

# Grouped Command List

- '{ }' runs in the current shell
- '( )' runs in a child/sub shell

# Grouped Command List Details

```
$ (sleep 3 ; sleep 5 ; sleep 8)
```

```
user 6299 6283 0 09:29 ? 00:00:00 sshd: user@pts/170
user 6303 6299 0 09:29 pts/170 00:00:00 _ -bash
user 5040 6303 0 16:33 pts/170 00:00:00 _ -bash
user 5100 5040 0 16:33 pts/170 00:00:00 _ sleep 8
```

```
$ { sleep 3 ; sleep 5 ; sleep 8 ; }
```

```
user 6299 6283 0 09:29 ? 00:00:00 sshd: user@pts/170
user 6303 6299 0 09:29 pts/170 00:00:00 _ -bash
user 6980 6303 0 16:35 pts/170 00:00:00 _ sleep 8
```

# Grouped Command List Details

```
$ (sleep 3 & sleep 5 & sleep 8 &)
```

```
user 6299 6283 0 09:29 ? 00:00:00 sshd: user@pts/170
user 6303 6299 0 09:29 pts/170 00:00:00 _ -bash
user 7891 1 0 16:37 pts/170 00:00:00 sleep 8
user 7890 1 0 16:37 pts/170 00:00:00 sleep 5
user 7889 1 0 16:37 pts/170 00:00:00 sleep 3
```

```
$ { sleep 3 & sleep 5 & sleep 8 & }
```

```
user 6299 6283 0 09:29 ? 00:00:00 sshd: user@pts/170
user 6303 6299 0 09:29 pts/170 00:00:00 _ -bash
user 9165 6303 0 16:39 pts/170 00:00:00 _ sleep 3
user 9166 6303 0 16:39 pts/170 00:00:00 _ sleep 5
user 9167 6303 0 16:39 pts/170 00:00:00 _ sleep 8
```

# Grouped Command List Details

- Grouped command list run in background are identical to '( )'

```
$ (sleep 3 ; sleep 5 ; sleep 8) &
```

```
$ { sleep 3 ; sleep 5 ; sleep 8 ; } &
```

```
user 6299 6283 0 09:29 ? 00:00:00 sshd: user@pts/170
user 6303 6299 0 09:29 pts/170 00:00:00 _ -bash
user 17643 6303 0 16:50 pts/170 00:00:00 _ -bash
user 17696 17643 0 16:50 pts/170 00:00:00 _ sleep 8
```

# Conditional Command List

- A list can execute sequence pipelines conditionally
- Execute cmd2 if cmd1 was successful

```
$ run_true && run_false
```

- Execute cmd2 if cmd1 was *NOT* successful

```
$ run_false || run_true
```

- Conditional lists can be grouped using single parentheses:

```
$ (run_random || (run_true && run_false))
```



# CONDITIONAL STATEMENTS

```
if .. elif .. else .. fi
```

```
if test-commands
then consequent-commands
elif more-test-commands
then more-consequents
else alternate-consequents
fi
```

```
if .. elif .. else .. fi
```

```
if test-commands ; then
 consequent-commands
elif more-test-commands ; then
 more-consequents
else
 alternate-consequents
fi
```

# Conditionals - if

- if .. then .. elif .. else .. fi

```
$ if [-e file] ; then echo file exists ; fi
```

- '[' is a builtin, equivalent to test

```
$ if test -e file ; then echo file exists ; fi
```

- Very similar to conditional list

```
$ [-e output] && echo file exists
```

- '[[ [ ] ]]' is extended test command

```
$ [[-e output]] && echo file exists
```

# Conditionals - if

- Must be a space between test statement and brackets

```
$ if [[-e file]] ; then echo file exists ; fi
-bash: [[-e: command not found
$ if [[-e file]] ; then echo file exists; fi
-bash: syntax error in conditional expression: unexpected
token `;'
-bash: syntax error near `;'
```

# Booleans for Math

- Can use math with conditionals in multiple tests

```
$ a=4
$ b=8
$ if ((($a % 4) == 0)) && ((($b % 4) == 0))
> then echo yes
> fi
yes
$
$ if ((($a % 8) == 0)) && ((($b % 8) == 0))
> then echo what
> fi
$
```

# Conditionals - test

- `test` has many different primaries and operators

```
$ help test
```

- Can be used for files (directories) or comparing strings and variables

# Pattern Matching Conditionals

- Test if a string matches a pattern
- Is a variable a number?

```
$ a=939
$ [[$a =~ "^[0-9]+$"]] && echo is a number
is a number
```

- Does a string end with an alphabetical character?

```
$ a=939
$ if [[$a = *[[:alpha:]]*]] ; then echo yes ; else echo
no ; fi
no
```

# Conditionals and exit status

- `if` and `test` use the exit status of the evaluated statement

```
$ test -d /tmp
$ echo $?
0
$ test -f silly
$ echo $?
1
$ [-s nonsense]
$ echo ?
1
```

# General if statements

- if evaluates exit status, so it can be used with any command

```
$ if grep -q bashrc ~/.bash_profile ; then echo yes ; fi
```

- The inverse is possible with ‘!’

```
$ if ! cat /zzz &> /dev/null ; then echo empty ; fi
```

- Identical to grouped command list

```
$ test -e /tmp && echo "/tmp exists!"
/tmp exists!
$ test $PATH == $HOME || echo "not equal"
not equal
```

# Unary vs. Binary

- Unary for testing files

```
$ if [-d /home/user/myDir] ; then touch cd myDir ; fi
```

- Binary for comparing variables

```
$ x=`grep -c string /proc/cpuinfo`
$ y=4
$ if [$x -eq $y]
> then echo "four times"
> fi
$
```

# Boolean Operators

- Multiple if statements in series

```
$ if [-e file1] && [-e file2] ; then echo both ; fi
```

```
case .. esac

case word in
 pattern)
 consequent-commands ;;
 more patterns)
 more-test-commands ;;
esac
```

# Conditionals - case

- `case .. esac` uses pattern matching

```
$ case `date +%a` in
Mon | T?? | Wed | Fri) echo "Weekday" ;; \
S*) echo "Weekend" ;; \
*) echo "Something Else" ;;
esac
Weekday
$
$ case `date +%A` in
Mon | T?? | Wed | Fri) echo "Weekday" ;; \
S*) echo "Weekend" ;; \
*) echo "Something Else" ;;
esac
Something Else
```



**LOOPS**

```
for .. do .. done

for name [in words]
do
 commands
done

for ((expr1 ; expr2 ; expr3))
do
 commands
done
```

# Loops - for

- **for** is used to step through multiple words

```
$ for i in moe larry curly ; do echo $i ; done
moe
larry
curly
```

- imitates **until** and **while** using **seq**:

```
$ for i in `seq 1 5` ; do echo $i ; done
1
2
3
4
5
```

# C-style for loop

- `for` can be used for integer traversal

```
$ for ((i=0 ; i < 10 ; i++))
> do
> echo $i
> done
0
1
2
3
4
5
6
7
8
9
```

`while .. do .. done`

```
while test-commands
do
 consequent-commands
done
```

until .. do .. done

until *test-commands*

do

*consequent-commands*

done

# Loops - until

- Until uses test commands
- Handy with math

```
$ a=1 ; until [[$a -gt 5]] ; do echo $a ; let a++ ;
done
1
2
3
4
5
```

- Can be used with semaphore files

```
$ until [[-e stop.file]] ; do sleep 60 ; done
```

# Loops - while

- `while` is the reverse of `until`

```
$ a=1 ; while [[$a -le 5]] ; do echo $a ; let a++ ;
done
1
2
3
4
5
```

# break And continue

- Can be used to end loops or skip sections

```
$ while [[$a -le 5]] ; do
> echo $a
> if ((a == 3)) ; then break ; fi
> let a++
> done
0
1
2
3
```

# Using while and read in a script

- `read` accepts a line from STDIN

```
while read var
do
 if [[$var == "exit"]]
 then
 break
 fi
 echo $var
 # do something else with $var
done
```

# Using for and case in a script

- Commandline argument parser:

```
arg=($@)
for ((i=0 ; i < ${#arg[@]} ; i++))
do
 let j=i+1
 case ${arg[$i]} in
 --help) halp=1 ;;
 --nonsense) nonsense=1 ;;
 --extra-juicy) extra=1 ;;
 esac
done

[[$halp]] && echo "Help? What help?" && exit
echo "Now for something completely different"
```



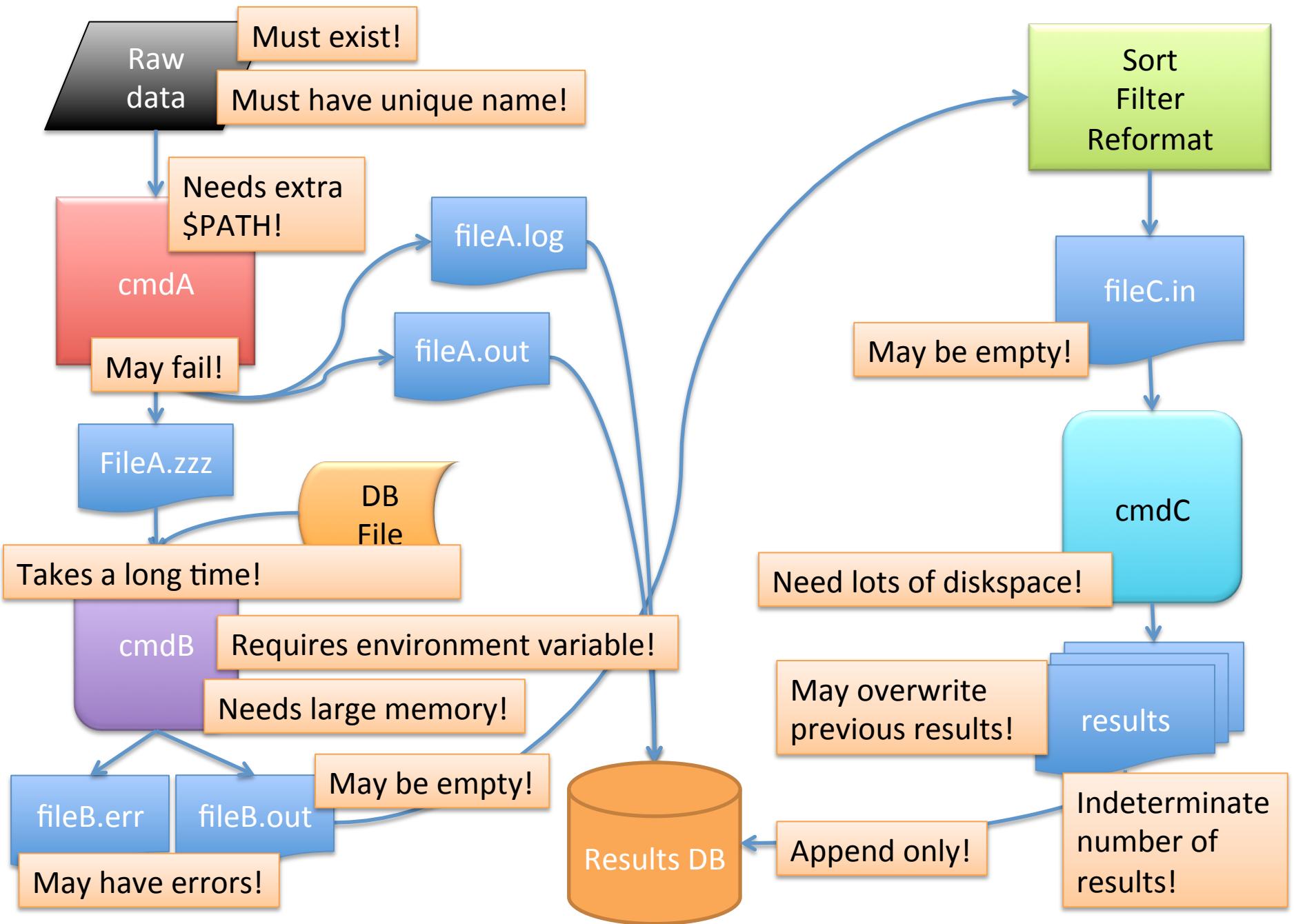
# FANTASY WORKFLOW SCRIPT

# Bash Scripting Tarball

- Download from:

<http://helix.nih.gov/Documentation/Talks/BashScripting.tgz>

```
$ ls -1
Empty_Raw_Data.txt
Proper_Unix_File.txt
Stupid Windows Or Mac Name.txt
cmdA
cmdB
cmdC
fantasy.sh
$./fantasy.sh rawdata name
```





# STUPID PET TRICKS

# Here Documents

- For multiline input

```
$ cat << EOF > file.txt
> This is a multiline
> input for something
> else.
> EOF
$
$ cat file.txt
This is a multiline
input for something
else.
$
```

- String can be anything, not just EOF.
- Must be against left margin (no initial spaces).

# Aliases Within Scripts

- Aliases have limited use within a script
- Aliases are not expanded within a script by default, requires special option setting:

```
$ shopt -s expand_aliases
```

- Worse, aliases are not expanded within conditional statements, loops, or functions

# Arrays

- Arrays are indexed by integers (0,1,2,...)

```
$ array=(moe larry curly)
```

- Arrays are referenced with {} and []

```
$ echo ${array[*]}\nmoe larry curly\n$ echo ${array[2]}\ncurly
```

- Can be used in for loops:

```
$ for i in ${array[*]} ; do echo $i ; done\nmoe\nlarry\ncurly
```

# Using Arrays

- The number of elements in an array

```
$ num=${#array[@]}
```

- Walk an array by index

```
$ for ((i=0 ; i < ${#array[@]} ; i++))
> do
> echo ${array[$i]}
> done
```

# Extended Math

- Use bc instead:

```
$ number=$(echo "scale=4; 17/7" | bc)
$ echo $number
2.4285
$ x=8
$ y=3
$ z=$(echo "scale = 3; $x/$y" | bc)
$ echo $z
2.666
```

- Might as well use perl or python instead...

# More Parameter Expansions

- Substrings

```
$ myvar="all cows eat grass"
$ echo ${myvar:0:3}
all
$ echo ${myvar:9:12}
eat grass
```

- Show all variables that match a prefix

```
$ echo ${!HO*}
HOME HOSTNAME HOSTTYPE
```

# More Parameter Expansions

- Set a default value

```
$ myvar="all cows eat grass"
echo ${myvar:-DEFAULT}
all cows eat grass
$ echo ${nothing:-DEFAULT}
DEFAULT
$ echo $nothing
```

- Make it stick

```
$ echo ${nothing:=DEFAULT}
DEFAULT
$ echo $nothing
DEFAULT
```



# SHELL OPTIONS

# bind

- bind sets and displays Readline keybindings
- Enables fine-tuning for commandline manipulation
- bind -v shows current settings
- ~/.inputrc file for permanency

# set

- Changes and displays shell options and variables
- Independent of environment

```
$ set -o
$ help set
```

- Useful ones: -x (xtrace), -v (verbose), -C (noclobber)

# set

- To turn on an option:

```
$ set -o noclobber
```

- To turn off an option:

```
$ set +o noclobber
```

# shopt

- Even more shell options
- Includes those from `set -o`
- Type the command '`shopt`' to see all available options
- Changes must be set in `~/.bashrc` or `~/.bash_profile` to be permanent

[http://wiki.bash-hackers.org/internals/shell\\_options](http://wiki.bash-hackers.org/internals/shell_options)



# EXTRAS

# Expanded List of Linux Commands

|        |           |        |          |          |        |         |          |
|--------|-----------|--------|----------|----------|--------|---------|----------|
| arch   | crontab   | emacs  | grep     | man      | ps     | split   | unexpand |
| at     | csplit    | env    | groups   | mkdir    | pwd    | ssh     | uniq     |
| awk    | cut       | ex     | gunzip   | mkfifo   | quota  | strings | unzip    |
| bc     | date      | expand | gzip     | mknod    | rcp    | sum     | users    |
| cal    | dc        | expr   | head     | more     | rename | tac     | vi       |
| cat    | dd        | factor | hostname | mv       | rlogin | tail    | watch    |
| cd     | df        | false  | id       | nano     | rm     | tar     | wc       |
| chgrp  | diff      | fgrep  | info     | nice     | rmdir  | tee     | whereis  |
| chmod  | diff3     | file   | install  | nl       | rsync  | test    | which    |
| chown  | dir       | find   | join     | nohup    | scp    | time    | who      |
| chroot | dircolors | finger | kill     | passwd   | screen | top     | whoami   |
| cksum  | dirname   | fmt    | less     | paste    | sdiff  | touch   | xargs    |
| clear  | du        | fold   | ln       | pathchk  | sed    | tr      | xdiff    |
| cmp    | echo      | free   | login    | pico     | seq    | true    | yes      |
| comm   | ed        | ftp    | logname  | printenv | sleep  | tty     | zcat     |
| cp     | egrep     | gawk   | ls       | printf   | sort   | uname   | zip      |

# Parameter Expansion

- How many characters?

```
$ echo ${#name}
7
```

- Substitute a pattern (not permanent)

```
$ echo ${name/er/rous}
monstrous
```

# Parameter Expansion

- Test if variable is defined

```
$ echo ${name:-other}
monster
$ echo ${names:-other}
other
```

- If undefined, set default value

```
$ echo ${names:=other}
other
$ echo $names
other
```

# Parameter Expansion

- Complain if the variable is undefined

```
$ echo ${bogus?Variable not defined}
-bash: bogus: Variable not defined
```

- Remove substring

```
$ file=/data/user/myfile.txt
$ echo ${file%.txt}
/data/user/myfile
$ echo ${file#/data*}
/usr/myfile.txt
```

# Setting Your Prompt

- The \$PS1 variable (primary prompt, \$PS2 and \$PS3 are for other things)
- Has its own format rules
- Example: PS1="[\u@\\h \w]\$ "

```
[user@host myDir]$ echo Hello World!
Hello World!
[user@host myDir]$
```

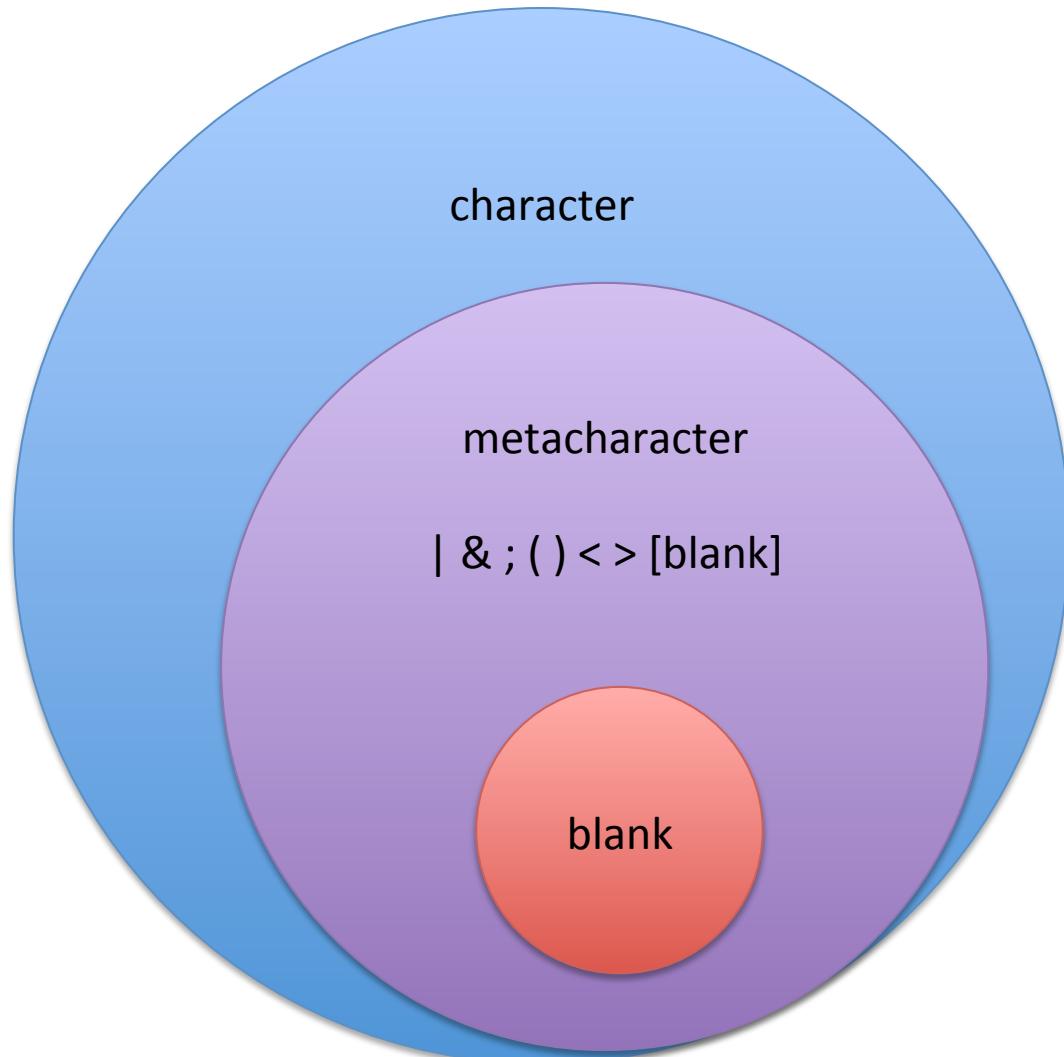
# Setting Your Prompt

- \d date ("Tue May 6")
- \h hostname ("helix")
- \j number of jobs
- \u username
- \W basename of \$PWD
- \a bell character (why?)

<http://www.gnu.org/software/bash/manual/bashref.html#Printing-a-Prompt>

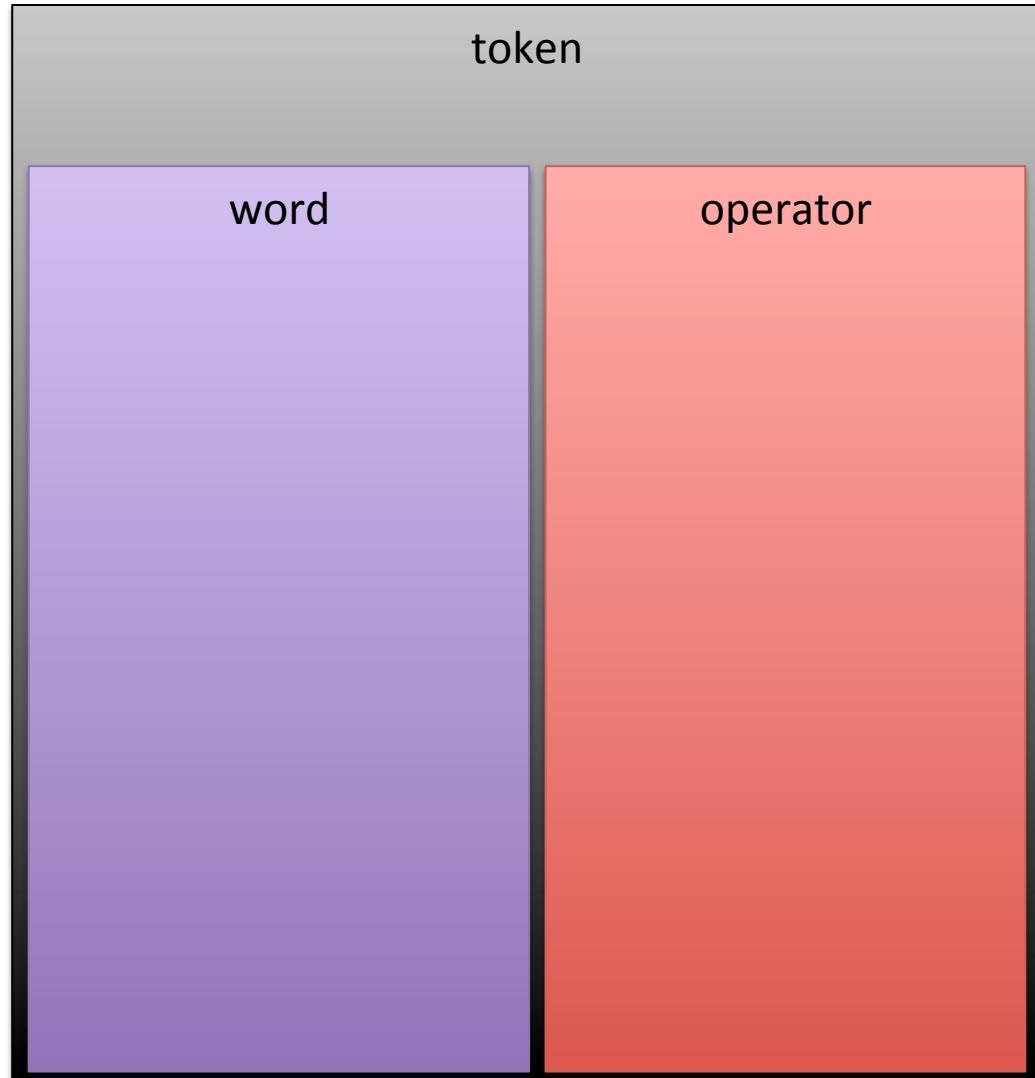
# Definitions

- Character: result of a keystroke
- Metacharacter: separates words
- Blank: space, tab, or newline



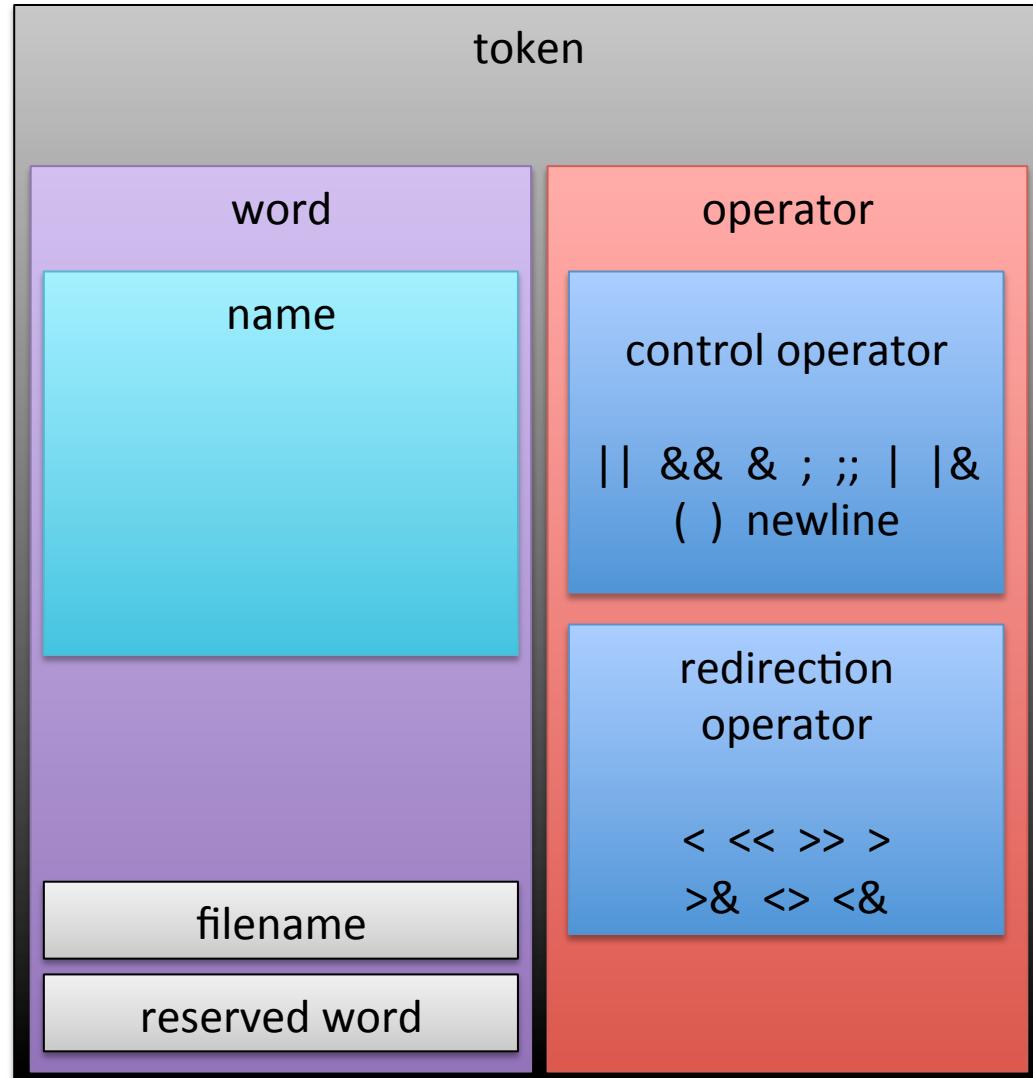
# Definitions

- Token: one or more characters separated into fields
- Word: a token with no unquoted metacharacters
- Operator: a token with one or more metacharacters



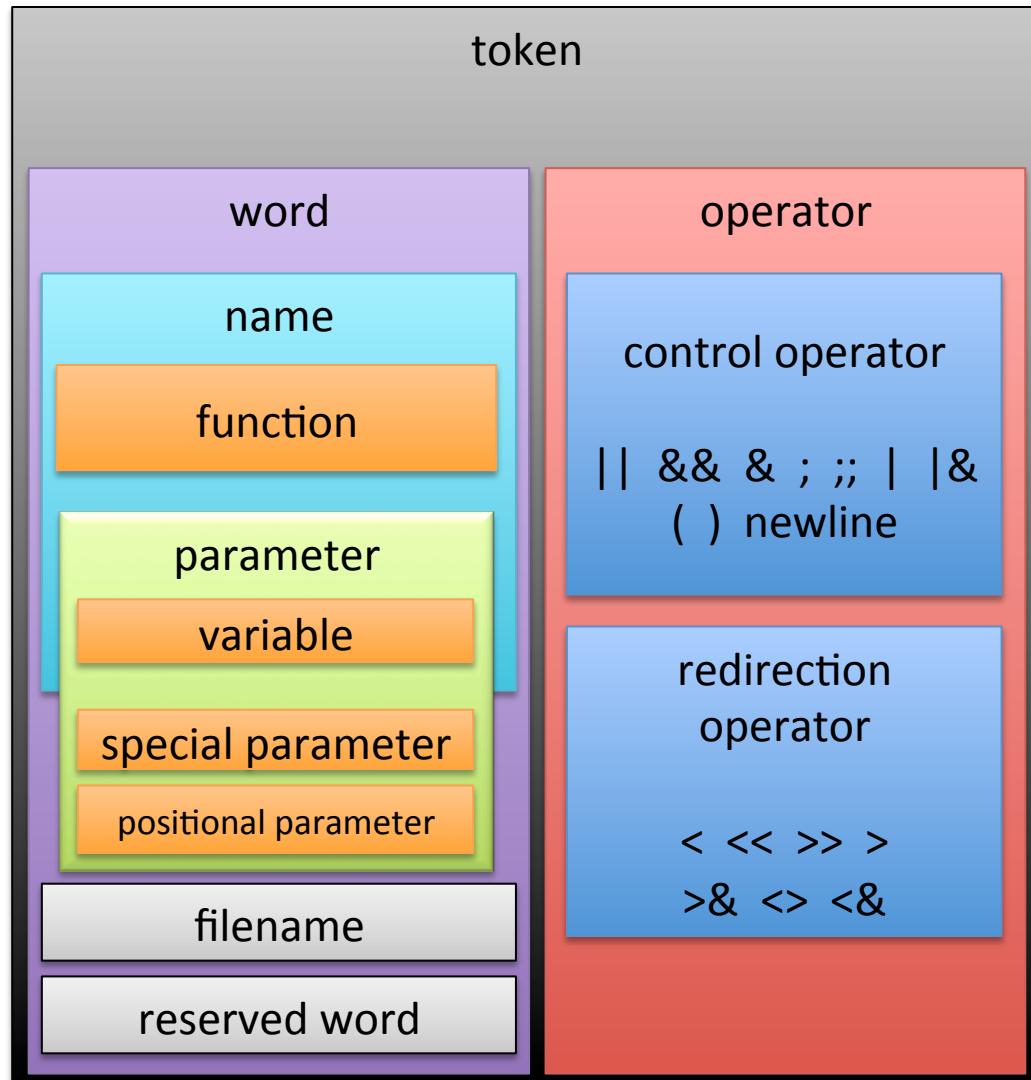
# Definitions

- Name: a word with [a..z,A..Z,0..9,\_] only
- Control operator: performs control functions
- Redirection operator: redirects file descriptors
- Filename: identifies a file
- Reserved word: special meaning , like **for** or **while**



# Definitions

- Parameter: stores values
- Variable: name for storing values, must begin with letter
- Special parameter: can't be modified
- Positional parameters: for retrieving arguments



# Questions? Comments?

**staff@helix.nih.gov**

